A New General Parser for Extensible Languages

Ian Lewis
School of Computing, University of Tasmania
GPO Box 252-100, Hobart, Tasmania,
Australia, 7001
ij_lewis@utas.edu.au

Abstract

This paper introduces a flexible parser that is capable of parsing any context-free grammars — even ambiguous ones. The parser is capable of arbitrary speculation and will consider all possible parses. This general parser is of particular use for the parsing of extensible languages as it allows macro programmers to forget about parsing, and concentrate on defining new syntax.

Keywords

Parsing, extensiblility, and programming languages.

1 Introduction

Programming languages that have compile-time meta-programming facilities range from rudimentary text-based macros such as those of C/C++ [X98], to syntax-driven macros such as those of Lisp [Ste90], MS^2 [WC93], and JSE [BP01].

Further to these, extensible programming languages such as Maya [Bak01] and Genesis [Lew05] allow the programmer to perform compile-time meta-programming and also to create entirely new syntactic forms.

Such extensible languages are relatively rare, as the creation of new syntax is a double-edged sword: it greatly increases the usefulness of meta-programming but disallows many common parser techniques and optimisations. The grammar of an extensible language is not fixed and the programmer must be able to conveniently modify — perhaps even midway through the parse.

A modifiable grammar requires consideration of two issues: the level of parser knowledge that is required of the programmer in order to modify the grammar, and the ability to modify the grammar during a parse.

1.1 Usability

The primary basis for judging a parser's suitability for parsing an extensible language is whether the user must understand the parser. As programming languages are traditionally fixed entities without users modifying their syntax, it is typical to modify the grammar to suit the parser:

> "In practice, grammars are often first designed for naturalness and then adjusted by hand to conform to the requirements of an existing parsing method." [GJ90§3.6.4, pp. 75]

This is unacceptable for use in an extensible language. The macro programmer should be shielded from such awkward manipulations.

For more traditional languages, parser efficiency is premium amongst design issues and effort is spent in ensuring this above all other concerns. Indeed, efficiency concerns need only be addressed once as the grammar is unchanging.

> "… making [an efficient] parser for an arbitrary given grammar is 10% hard work; the other 90% can be done by computer." [GJ90§3.6.4, pp. 75]

We consider the sacrifice of efficiency essential in order to remove the burden of hard work from the macro programmer. Ideally, we would not need to make this choice, but as a starting point for an extensible language we should not unduly restrict our parser.

1.2 Mid-parse Grammar Modification

It is clear that an extensible parser must be capable of handling an augmented grammar and then use this grammar to parse a given input. In addition to this, extensible languages need a mechanism for specifying which modifications are to be made to the grammar. This is typically done with a mechanism similar to that used to import functions or classes from libraries.

For example, in Genesis, this mid-parse grammar modification the grammar occurs at the level of `import` statements contained in the file. The parsing of symbols beyond the `import` statement must be performed with the amended grammar. Maya has a special `use` keyword that modifies the grammar within the current scope.

Correct handling of such mid-parse grammar modifications is essential to the correct function of extensible languages. As will be shown, the modification of a grammar midway through a parse places proves to be a harder requirement to meet than reducing the burden on the programmer.

1.3 Overview

The remainder of this paper is presented as follows: In section 2, we provide a brief explanation of the concept of general parsing and look at two often used general parsers. In section 3, we present the development of a new general parser. In section 4, we present an technique for optimization of the final algorithm of section 3. In section 5, we perform an analysis of the performance of this new algorithm. In Section 6, we conclude the paper with a brief mention of further work.

2 General Parsing

General parsing allows a context-free grammar to be in any form. These parsers should at least be able to meet our first requirement for extensible language parsing — i.e. ease of user for the programmer.

We examine two general parsing mechanisms CYK and Earley parsing.

2.1 CYK Parsing

The CYK (named after its independent co-creators Cocke [CS70], Younger [You67], and Kasami [Kas65]) algorithm provides parsing of ambiguous grammars, but the standard version requires grammars to be in Chomsky Normal Form (CNF) [GJ90]. Context-free grammars can be converted to CNF without too much difficultly [GJ90], so CYK parsing still serves as a good starting point for a general parser. The standard algorithm can be extended to handle forms that are not CNF, but at the cost of a more difficult to implement algorithm.

CYK parsing considers all possible subsequences of the input string starting with those of length one, then two, etc. Once a rule is matched on a subsequence, the left-hand side is considered a possible valid replacement for the underlying subsequence.

Typically standard CYK is implemented using multidimensional Boolean arrays [GJ90]; each entry representing the successfulness of applying a rule to a subsequence (represented by a start index and a sequence length). An algorithm for standard CYK parsers using a multidimensional array is shown in Figure 2.1.

CYK parsers operate in a non-directional bottom-up fashion. They are non-directional as they match rules of a given length at all places in the input.

```
int N = /* number of input tokens */
int R = /* number of rules in CNF grammar */

bool array[N][N][R];

foreach token T at index I in the input
  foreach rule R -> T
    array[I][1][R] = true

foreach I = 2..N
  foreach J = 1..N-I+1
    foreach K = 1..I-1
      foreach rule R -> S T
        if P[J][K][S] and P[J+K][I-K][T]
          P[J][I][R] = true
```

Figure 2.1: CYK Algorithm

```
repeat until input is exhausted
  a = /* current input symbol */
  k = /* current state index */

  repeat until no more states can be added
    foreach state (X ::= A•YB, j) in state[k]        // prediction
      foreach rZule (Y ::= C)
        state[k].add( state (X ::= •C, k) )

    foreach state (X ::= A•aB, j) in state[k]        // scanning
      state[k+1].add( state (X ::= Aa•B, j) )

    foreach state (X ::= A•, j) in state[k]          // completion
      foreach state (Y ::= A•B, i) in state[j]
        state[k].add( state (Y ::= A•XB, i) )
```

Figure 2.2: Earley's Algorithm

2.2 Earley's Algorithm

Earley's algorithm [Ear70] is described as a breadth-first top-down parser with bottom-up recognition. The algorithm maintains a list of states which each contain a list of partially complete rules. These partially complete rules are written with a dot representing the currently examinable position in a rules right-hand side. For example, in `X::=ab•c` the terminals `a` and `b` have been examined and `c` is the next terminal to be examined.

At each stage in the parse, the following three actions occur in turn: prediction, scanning, and completion. At each iteration of the algorithm any of these actions may add a partially completed rule to either the current state or the next state. Each rule has associated with it a source state.

Prediction adds to the next state each of the rules for which a non-terminal appears immediately to the right of the most recently parsed symbol in that rule.

Scanning adds to the next state all partially complete rules that expect the current input symbol as their next symbol.

For each completed rule in the current set, completion adds to the current state all rules from the corresponding source state, that have most recently examined (i.e. in which the dot appears immediately to the right of) the entire right-hand side of the completed rule.

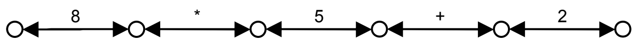
Figure 2.2 shows an algorithm for Earley parsing. In this algorithm, the symbols X and Y represent any non-terminal; and A, B, and C represent any sequence of symbols.

```
expr ::= expr + expr | expr * expr | ( expr ) | number
```

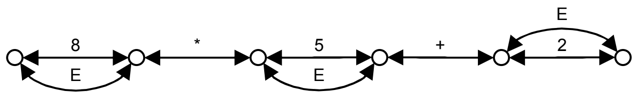
Figure 3.1: Ambiguous Simple Expression Grammar

```
repeat until no changes made
  foreach vertex V in original set
    foreach forward path (V, U) of length ≤ longest rule length
      foreach rule R in rule set of equal length to the path (V, U)
        if R's right-hand side matches path values
          add new edge from (V, U) with R's left-hand side as value
if there exists an edge from start vertex to end vertex the entire
  input has been recognised
```

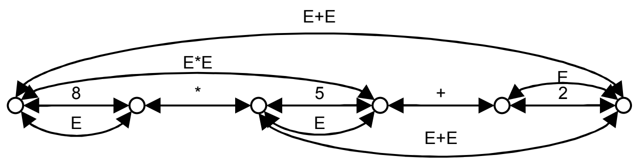
Figure 3.2: Multipass Graph Expansion Algorithm



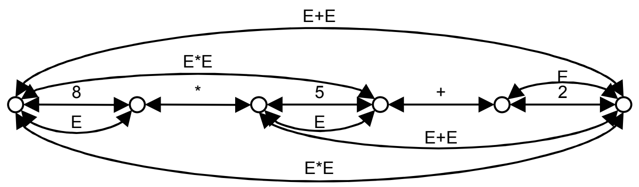
(a) Initial Input



(b) After Pass One



(c) After Pass Two



(d) After Pass Three and Four

Figure 3.3: Multipass Graph Expansion

2.3 Suitability for Extensible Languages

These general parsers at least partially satisfy our requirements for extensible language parsing, but are far from perfect.

Usability

CYK and Earley parsing do not suffer from the limitations of traditional top-down and bottom-up approaches to parsing. Such general parsers seem well suited to providing a simple to use system for end-users. A general parser should be capable of parsing any grammar the user creates.

A system built with such a parser must have facilities for resolving ambiguity and reporting when ambiguities are not resolved.

Mid-parse Grammar Modification

CYK parsing considers all substrings of length one up to the length of the input. This process is unsuitable if we do not know all of the rules in advance.

Earley parsing speculatively keeps track of partially accepted rules in a top-down fashion (even though the rules are accepted in a bottom-up order). If we introduce new rules via an import statement, then these speculations are incomplete.

2.4 Summary

Neither CYK nor Earley parsers satisfy all requirements for extensible programming languages. In order to allow for the grammar to be modified in the middle of a parse a new parser is required.

3 Development

Two initial algorithms were produced for graph expansion parsing. The first attempt was to test the feasibility of providing generality and as a result was very inefficient, but importantly, it met the goal of mid-parse grammar modification. The second algorithm was designed to have a stronger concept of completion.

In the next section we introduce an optimised version of the second algorithm from this section is introduced.

3.1 Multipass Method

The original technique for building a graph of all possible parses and subparses involved making a series of passes through the entire input string.

The algorithm begins with a graph of the input — in all examples in this section just a trivially linear graph, but the algorithm does not restrict the complexity of the input. Each vertex in the graph contains a forward edge that contain a single token of input —an example input for the grammar shown in Figure 3.1, the simple string `5*8+2`, is shown in Figure 3.3(a).

The algorithm iterates through each of the vertices in order. All forward paths (that are no longer than the longest right-hand side of any grammar rule) from each vertex have their edge values matched against all grammar rules of equal length. If a match is found, a new edge is added from the beginning to the end of the path with its value being the left-hand side of the grammar production.

One complete iteration through the vertices does not produce all of the possible rule reductions, so this process is repeated until no further additions are made to the graph. An algorithm for this multipass method is shown in Figure 3.2.

An example execution of this technique for the grammar of Figure 3.1 with the initial input `8*5+2` is shown in

Figure 3.3 (with `expr` abbreviated to `E`). Even for such a simple test case, the algorithm requires four entire passes of the input be completed.

There is one major problem with this technique. The multiple passes are inefficient, and each pass must consider an increasing number of path possibilities as the complexity of the graph increases. The final pass only serves to provide termination yet it takes the most time.

This algorithm succeeds in parsing arbitrary grammars and at mid-pass grammar modification. Though the multiple passes are inefficient, they do allow a newly modified grammar to be applied to the entire input.

The multipass method is a non-directional bottom-up parser. If restricted to paths of a fixed (but increasing) length for each pass, it acts as a generalised CYK parser.

3.2 Single Pass Method

The major problem with the multipass method is that for each vertex iteration in a pass, rules are being matched against paths that contain vertices that have not been examined in the current pass. This means matching has not been performed between the ruleset and the majority of the subpaths of the path currently being examined (i.e. all subpaths not containing the left-most vertex).

The idea behind the single pass method is to scan the vertices from left-to-right, but to consider the paths from the current vertex from right-to-left (i.e. backward paths not forward paths). This approach ensures that all subpaths have been fully compared against the rule set. It also ensures that once the last vertex is examined that all possible parses have been generated.

The single pass graph expansion algorithm is shown in Figure 3.4.

Figure 3.6 shows an example execution of the single pass technique for the grammar of Figure 3.1 with the initial input of `8*5+2`. The current node is highlighted at each step. The resultant graph is identical to that produced by the multipass technique shown in Figure 3.3 but it is produced in a more efficient fashion.

The paths that are examined by the algorithm during the production of Figure 3.6 and when new edges are added to the graph are demonstrated in Figure 3.5. The only non-terminal in the grammar of Figure is for that of `expr`, so each time a new expression is found it is given a subscript so that the process is easier to follow. Each generated path is no longer than the longest rule in the grammar and must be compared to all grammar rules for a match.

The single pass method is left-to-right scanning bottom-up parser. On non-ambiguous grammars it will produce a single right-derivation traced out in reverse in a similar fashion to a LR parser.

```
foreach vertex V in original set
  foreach backward path (U, V) of length ≤ longest rule length
    foreach rule R in rule set of equal length to path (U, V)
      if R's right-hand side matches path values
        add new edge from (U, V) with R's left-hand side as value

if there exists an edge from start vertex to end vertex the entire
  input has been recognised
```

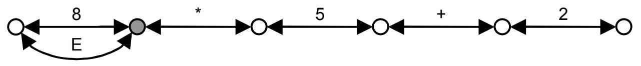
Figure 3.4: Single Pass Graph Expansion Algorithm

Iteration / Vertex	Examined Path	Action
0	()	
1	(8)	→ E_1 on edge (0, 1)
	(E_1)	
2	(*)	
	(8, *)	
	(E_1, *)	
3	(5)	→ E_2 on edge (2, 3)
	(*, 5)	
	(8, *, 5)	
	(E_1,*, 5)	
	(E_2)	
	(*, E_2)	
	(8, *, E_2)	
	(E_1, *, E_2)	→ E_3 on edge (0, 3)
	(E_3)	
4	(+)	
	(5, +)	
	(*, 5, +)	
	(E_2, +)	
	(*, E_2, +)	
	(E_3, +)	
5	(2)	→ E_4 on edge (4, 5)
	(+, 2)	
	(5, +, 2)	
	(E_2, +, 2)	
	(E_3, +, 2)	
	(E_4)	
	(+, E_4)	
	(E_2, +, E_4)	→ E_5 on edge (2, 5)
	(E_3, +, E_4)	→ E_6 on edge (0, 5)
	(E_5)	
	(*, E_5)	
	(8, *, E_5)	
	(E_1, *, E_5)	→ E_7 on edge (0, 5)
	(E_6)	
	(E_7)	

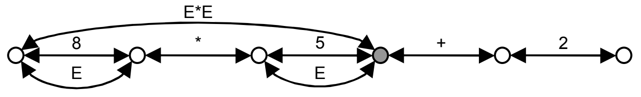
Figure 3.5: Single Pass Graph Expansion Evaluation

4 Optimisation

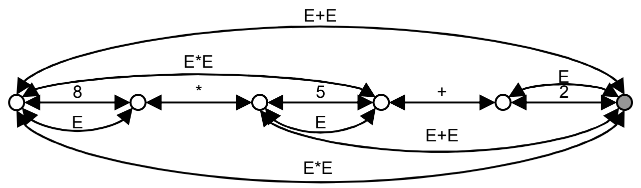
Many of the paths examined in Figure 3.5 are clearly not going to match any of the rules of the simple expression grammar. For example, no rule ends with a + symbol yet in iteration 4 we examined six paths that end with a + symbol.



(a) After Iteration One



(b) After Iteration Three



(c) After Iteration Five

Figure 3.6: Single Pass Graph Expansion

```
parse
  foreach vertex V in original set
    check (V, V)
  if there exists an edge from start vertex to end vertex the
    entire input has been recognised

check (path (U, V))
  foreach backward edge (T, U)
    if path (T, V) matches a grammar rule R's right-hand side
      add a new edge from (T, V) with R's left-hand side as value
    if further possibilities end with this subsequence
      check (T, V)
```

Figure 4.1: Final Graph Expansion Algorithm

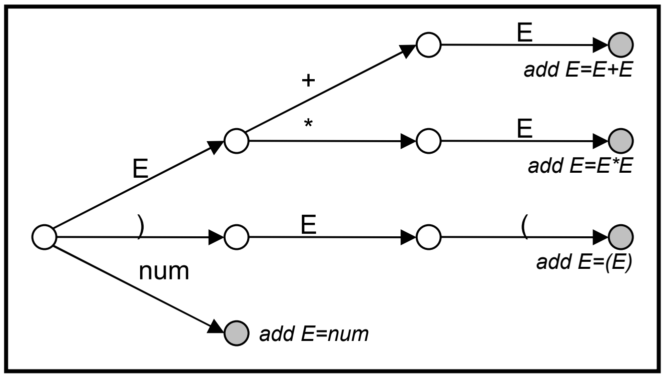


Figure 4.2: Simple Expression Partial Match Tree

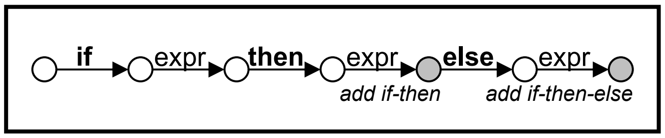


Figure 4.3: Dangling-else Partial Match Tree

The optimised algorithm does not continue to expand paths unnecessarily. It keeps track of how many possible rules contain the current path as the right-most part of their right-hand side, and when this falls to zero the current path is abandoned. Its execution produces the same order of graph additions as the single pass method of the previous subsection.

Iteration / Vertex	Examined Path	Action
0	()	
1	(8)	→ E_1 on edge (0, 1)
	(E_1)	
2	(*)	
3	(5)	→ E_2 on edge (2, 3)
	(E_2)	
	(*, E_2)	
	(8, *, E_2)	
	(E_1, *, E_2)	→ E_3 on edge (0, 3)
	(E_3)	
4	(+)	
5	(2)	→ E_4 on edge (4, 5)
	(E_4)	
	(+, E_4)	
	(E_2, +, E_4)	→ E_5 on edge (2, 5)
	(E_3, +, E_4)	→ E_6 on edge (0, 5)
	(E_5)	
	(*, E_5)	
	(8, *, E_5)	
	(E_1, *, E_5)	→ E_7 on edge (0, 5)
	(E_6)	
	(E_7)	

Figure 4.4: Final Algorithm Graph Expansion Evaluation

Figure 4.1 contains an algorithm for this optimised method. Paths are generated incrementally by the recursive algorithm `check`.

The difficult part of this algorithm is determining whether the current path matches the right subsection of the right-hand side of a rule (or set of rules). To aid in this process a tree of partial matches is used. Figure 4.2 contains this tree of partial matches for the simple expression grammar of Figure 3.1. The arcs show the matched tokens, and the nodes contain the addition rules that represent the reduction of rules.

Although this tree contains add actions only at the nodes of the tree this is not typical of more complicated grammars. These add actions may appear at any node in the tree. For example, with a right-to-left parse of the classic "dangling-else" grammar both the `if-then` statement and the `if-then-else` statement are contained in a single path in the tree (see Figure 4.3). A more contrived left-to-right example would be for a grammar containing bracketed expressions and single parameter procedure calls.

Figure 4.4 shows the paths examined by the final Graph Expansion Parser for the simple expression grammar with input `8*5+2`. These are a strict subset of those in Figure 3.5.

5 Analysis

Graph Expansion Parsing was designed specifically for the implementation of Genesis, and in this section we examine its performance against the general parsers of Earley's algorithm and the CYK parser.

5.1 Acceptable Grammars

Graph Expansion Parsing can operate on any context-free grammars without empty symbols. This class of grammars is far larger than those than can be accepted by CYK, but smaller than Earley's algorithm which allows empty symbols.

The lack of empty symbols does not overly restrict the languages that can be accepted; it is an easy process to remove empty symbols and while the result is more verbose but no less understandable.

5.2 Efficiency

In this section, the efficiency of Graph Expansion Parsing is compared theoretically against the general parsers of both Earley and CYK. Also, empirical results are compared to Earley's algorithm with the same set of tests as his original paper [Ear70]. In most tests, Graph Expansion Parsing performs on par with the Earley parser.

Given `n` input tokens, both Earley and CYK parsers require at worst $O(n^3)$ time. However, $O(n^3)$ is a requirement for CYK but merely an upper bound for Earley. On *bounded state grammars* [Ear70] (this includes most LR(k) grammars) Earley's algorithm operates in linear time. Earley describes three grammars which generate similar languages (shown in Figure 5.1) that take $O(n)$, $O(n^2)$, and $O(n^3)$ time respectively.

GEP has worst case time complexity of $O(n^3)$, but like Earley's algorithm, it can perform with better complexities on certain grammars. GEP operates on the grammars of Figure 5.1 in $O(n^2)$, $O(n^3)$, and $O(n^3)$ time respectively.

Given `n` input tokens, both Earley and CYK parsers require $O(n^2)$ space. However, $O(n^2)$ is an upper bound for Earley but a requirement for CYK. These complexities are for recognising a given string, not for producing all possible parse trees. For example, the grammar of Figure 5.1(c) produces an exponential number of possible parses for a given input string, so any algorithm that provides all such parses can do no better than $O(2^n)$ space complexity.

Similarly, the space requirements of Graph Expansion Parsing are dependent upon how ambiguity is handled. If ambiguities are fully resolved as the parse progresses then the space requirements are bounded by $O(n^2)$, if not, the bound is $O(2^n)$.

Empirical Results

Earley compares his algorithm with a variety of backtracking techniques [Ear70]. It is clearly shown that

```
K ::= J | K J
J ::= F | I
F ::= x
I ::= x
```

(a) Earley O(n) Grammar

```
A ::= x | x A x
```

(b) Earley $O(n^2)$ Grammar

```
A ::= x | A A
```

(c) Earley $O(n^3)$ Grammar

Figure 5.1: Time Complexities of Earley's Algorithm Theoretical Performance

Grammar	Sentence	Earley	GEP LR	GEP RR	GEP adds
S=Ab A=a|Ab	ab^n	4n+7	6n+1	6n+1	2n+1
S=aB B=aB|b	a^{n}b	6n+4	6n+1	6n+1	2n+1
S=ab|aSb	a^{n}b^n	6n+4	7n-3	7n-3	n
S=AB A=a|Ab B=bc|bB|Bd	ab^{n}cd	18n+8	14n+7	14n+6	8n-3

Table 5.1: Earley Versus GEP Time Complexity

Grammar	Sentence	Earley	GEP LR	GEP RR	GEP adds
X=a|Xb|Ya	ededea	33	37	35	11
Y=e|YdY	ededeab^4	45	77	98	27
	ededeab^{10}	63	137	188	51
	ededeab^{200}	633	2037	3038	811
	(ed)^{6}eabb	79	123	152	43
	(ed)^{7}eabb	194	292	363	119
	(ed)^{8}eabb	251	371	460	159
S=AB	adbcddb	44	35	35	13
A=a|SC	ad^{3}bcbcd^{3}bcd^{4}b	108	97	97	35
B=b|DB	adbcd^{2}bcd^{5}bcd^{3}b	114	82	82	30
C=c	ad^{18}b	123	115	115	39
D=d	a(bc)^{3}d^3(bcd)^{2}dbcd^{4}b	141	127	127	47
	a(bcd)^{2}dbcd^{3}bcb	95	83	83	31
F=C|S|P|U	p	28	2	2	1
C=U⊃U	(pΛq)	68	23	25	4
U=(F)|~U|L	(p'Λq)VrVpVq'	148	66	95	18
L=L'|p|q|r	p⊃((q⊃~(r'V(pΛq)))⊃(q'Vr)	277	90	151	21
S=UVS|UVU	~(~p'Λ(qVr) Λp'))	141	59	129	21
P=UΛP|UΛU	((pΛq)V(qΛr)V(rΛp'))⊃ ~((p'Vq')Λ(r'Vp))	399	143	236	34

Table 5.2: Earley Versus GEP Comparison

his algorithm is superior to other general parsers. Graph Expansion Parsing was compared with Earley parsing on all of these grammars.

All of the following time complexities are calculated based on *primitive operations*. For Earley's method, the primitive operation used is the act of adding a state to the state set, and for GEP it is attempted matching of a path. GEP paths are built incrementally so each check is effectively a constant operation.

In Table 5.1 the time complexities of Earley parsing and GEP are compared. Shown for GEP is both a forward and backwards scan of the input and also the number of edges added to the graph. The first three grammars compared demonstrate left-, right-, and centre-recursive forms respectively. The fourth grammar effectively contains all three recursive forms.

Both Earley's method and GEP parse all these grammars in linear time, although GEP generally has a smaller constant factor than Earley's method. No significant difference is seen with GEP between scanning the input left-to-right or right-to-left.

Table 5.2 compares Earley parsing and GEP on more complicated grammars with mutually recursive components. The third grammar is the most representative of a real programming language grammar. The choice of strings is taken from [Ear70] so that a direct comparison could be made.

Graph Expansion Parsing performs the most favourably on the third grammar which is a representation of a propositional calculus. As this is the most "real world" of the grammars, GEP seems well suited to non-theoretic use.

With two of these three grammars sizeable differences are visible between performing a left-to-right scan of the input to performing a right-to-left scan. In general the left-to-right scan performs considerably better. The largest difference is in the first grammar and is due to the predominance of left-recursive elements.

6 Conclusion

This paper demonstrates an efficient general parsing technique that compares well to both Earley and CYK parsers. The final parser is particularly well-suited to extensible programming languages — in fact, the Genesis[1] programming language relies on GEP for its implementation.

6.1 Planned Future Work

Section 4 already detailed some optimisations to the parser, but nonetheless the current Graph Expansion Parser has much room for improved performance.

It may prove to be possible to discover sub-graphs that have no possibility of further additions and such forms could be ignored for the rest of the parse. The current algorithm performs many checks that are required and does so repeatedly. Any graph pruning technique would provide quite a boost in efficiency.

Another minor improvement could come from collapsing some of the information in the partial match tree. A simple example of this kind of operation is if a grammar contains a rule that converts a token into an identifier and a rule that converts an identifier into a simple expression then upon successful conversion of the token into an identifier we can produce an expression simultaneously without further matching. It may even be possible to apply this approach in a more general way to improve efficiency.

[1] The GEP implementation for the Genesis language can actually handle context-sensitive grammars as well, as each accepting macro may choose to fail if further specified conditions are not met.

References

[Bak01] Jason Baker: *Macros that Play: Migrating from Java to Maya*, Master's Thesis, University of Utah, December 2001.

[BP01] Jonathan Bachrach and Keith Playford: The Java Syntactic Extender (JSE). In, *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 31–42, Tampa Bay, Florida, 2001.

[CS70] John Cocke and Jacob T. Schwartz: *Programming languages and their compilers: Preliminary notes*, Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[GJ90] Dick Grune and Ceriel Jacobs: *Parsing Techniques, A Practical Guide*, Ellis Horwood, Chichester, West Sussex, 1990. ISBN 0-13-651431-6.

[Kas65] T. Kasami: *An efficient recognition and syntax-analysis algorithm for context-free languages*, Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, Massachusetts, 1965.

[Lew05] Ian Lewis: *Genesis: An Extensible Java*, PhD Thesis, University of Tasmania, December 2005.

[Ste90] G.L. Steele Jr., *Common Lisp: The Language*, 2nd edition, Bedford, Massachusetts, Digital Press, 1990.

[WC93] Daniel Weise and Roger Crew: Programmable Syntax Macros. In, *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI '93)*, pp. 156–165, Albuquerque, New Mexico, June 1993.

[You67] Daniel H. Younger: *Recognition and parsing of context-free languages in time n^3*, Information and Control, Volume 10, Number 2, pp. 189–208, 1967.

[X98] X3 Secretariat. International Standard – The C++ Language. X3J16-14882. *Information Technology Council (NSITC), 1998.*